Design and Modeling of a Non-blocking Checkpointing System

Kento Sato
Dep. of Mathematical and Computing Science
Tokyo Institute of Technology
2-12-1-W8-33, Ohokayama,
Meguro-ku, Tokyo 152-8552 Japan
Email: kent@matsulab.is.titech.ac.jp

Kathryn Mohror, Adam Moody,
Todd Gamblin and Bronis R. de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551 USA
Email: {kathryn, moody20, tgamblin, bronis}@llnl.gov

Naoya Maruyama
Advanced Institute for Computational Science
RIKEN
7-1-26, Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, 650-0047 Japan
Email: nmaruyama@riken.jp

Satoshi Matsuoka
Global Scientific Information and Computing Center
Tokyo Institute of Technology
2-12-1-W8-33, Ohokayama,
Meguro-ku, Tokyo 152-8552 Japan
Email: matsu@is.titech.ac.jp

***Abstract*—As the capability and component count of systems increase, the MTBF decreases. Typically, applications tolerate failures with checkpoint/restart to a parallel file system (PFS). While simple, this approach can suffer from contention for PFS resources. Multi-level checkpointing is a promising solution. However, while multi-level checkpointing is successful on today's machines, it is not expected to be sufficient for exascale class machines, which are predicted to have orders of magnitude larger memory sizes and failure rates. Our solution combines the benefits of non-blocking and multi-level checkpointing. In this paper, we present the design of our system and model its performance. Our experiments show that our system can improve efficiency by 1.1 to 2.0× on future machines. Additionally, applications using our checkpointing system can achieve high efficiency even when using a PFS with lower bandwidth.**

***Keywords*-Fault tolerance; Checkpoint/Restart; Markov model;**

I. Introduction

The computational power of High Performance Computing (HPC) systems is growing exponentially, which enables finer grained scientific simulations. However, the overall failure rate of HPC systems increases with their size. For example, in the year and a half from November 1st 2010 to April 6th 2012, TSUBAME2.0, ranking 5th in the Top500 list [1] (November 2011), experienced 962 node failures ranging from memory errors to whole rack failures [2]. Thus, a failure occurred every 13.0 hours on average. Further, the MTBF (mean time between failure) of future systems is projected to shrink to tens of minutes [3]. Without a viable resilience strategy, applications will be unable to run for even one day on such a large machine. Thus, resilience in HPC has become more important than ever as we plan for future systems.

Checkpointing is an indispensable fault tolerance technique, commonly used by HPC applications that run continuously for hours or days at a time. A *checkpoint* is a snapshot of application state that can be used to restart execution if a failure occurs. However, when checkpointing large-scale systems, tens of thousands of compute nodes write checkpoints to a parallel file system (PFS) concurrently, and the low I/O throughput becomes a bottleneck. Although simple, this straightforward checkpointing scheme can impose huge overheads on application run times.

Multi-level checkpointing [4], [5] is a promising approach for addressing these problems. This approach uses multiple storage levels, such as RAM, local disk, and the PFS, according to the different degrees of resiliency and the cost of checkpointing in those storage levels. Multi-level checkpointing systems typically rely on node-local storage levels for restarting from more common failures, such as single-node failures, and the PFS for more catastrophic failures. By taking frequent, inexpensive node-local checkpoints, and less frequent, high-cost checkpoints to the PFS, applications can achieve both high resilience and better efficiency.

However, computational capabilities are increasing faster than PFS bandwidths. This imbalance in performance means applications can be blocked for tens of minutes for a single checkpoint [4]. Thus, the overhead of checkpointing to the PFS can dominate overall application run time even with infrequent PFS checkpoints. Further, the huge numbers of concurrent I/O operations from large-scale jobs burden the PFS and are themselves a major source of failures. Thus, we must reduce the PFS load in order to achieve high reliability and efficiency.

Our non-blocking checkpointing system solves this problem through agents running on additional nodes that asynchronously transfer checkpoints from the compute nodes to the PFS. Our approach has two key advantages. It lowers application checkpoint overhead by overlapping computation and writing checkpoints to the PFS. Also, it reduces PFS load by using fewer concurrent writers and moderating the rate of PFS I/O operations. In particular, our non-blocking checkpointing

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

system maintains a given application efficiency with significantly lower PFS requirements than blocking checkpointing.

We make the following major contributions:

- The design of a non-blocking checkpointing system;
- A detailed failure analysis of a petascale supercomputer;
- A Markov model of non-blocking checkpointing on top of an existing multi-level checkpointing system;
- A comprehensive exploration of non-blocking checkpointing on current and future systems.

Our results show that combining non-blocking and multi-level checkpointing results in highly efficient application runs with low PFS bandwidth requirements.

II. The Scalable Checkpoint/Restart (SCR) Library

Our non-blocking checkpointing system is developed as an extension to the SCR library [6]. Our system asynchronously transfers node-local checkpoints written by SCR to the PFS.

A. Checkpoint/Restart Scheme

SCR is a checkpoint/restart library that production LLNL applications use. It supports globally-coordinated checkpoints that the application writes, primarily as a file per MPI process – a common checkpointing technique in large-scale codes. SCR uses hierarchically distributed storage to implement a multi-level checkpointing system. For instance, SCR can cache the most recent checkpoints in RAM disks or SSDs that are local to the compute nodes on which the application is running, and it can also store checkpoints on the PFS.

SCR applies redundancy schemes to the distributed storage devices such that the application can recover from most failures using the cached checkpoints. When these storage devices are local to and scale with the number of compute nodes, the cost to cache a checkpoint is low, so the application can checkpoint frequently. However, this approach cannot handle failures involving many nodes so SCR periodically copies (flushes) a cached checkpoint to the PFS in order to recover from those more severe, but less frequent failures. After a failure, SCR tries to restart from the most recent checkpoint in cache. If it cannot, it fetches and restarts from the most recent checkpoint on the PFS.

B. SCR Flush

As mentioned, to withstand catastrophic failures, such as a rack-level failure or a data center-wide power outage, SCR periodically flushes a checkpoint from node-local storage to the PFS. SCR supports two types of flush operations: synchronous and asynchronous. When SCR copies a checkpoint to the PFS synchronously, it blocks the application until the copy has completed. In large-scale computations on tens of thousands or more compute nodes, the total checkpoint size can reach multiple terabytes, which may take tens of minutes to complete. Thus, synchronous flushes can dominate application run time. SCR also supports an asynchronous flush in which it starts the flush and immediately returns control to the application. An external and independent process that runs in the background then copies the checkpoint to the PFS, so the application is not blocked during this transfer.

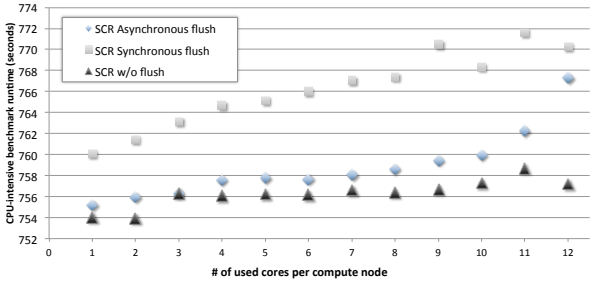

SCR Asynchronous flush
SCR Synchronous flush
SCR w/o flush
CPU-intensive benchmark runtime (seconds)
# of used cores per compute node

Fig. 1: IOR run time on a 12-core machine with SCR

TABLE I: TSUBAME2.0 Failure Category

FC	# of nodes affected	Failure points	Failure rate (failures/sec)	MTBF
5	1408	PFS, Core switch	0.1778×10^{-6}	65.10 days
4	32	Rack	0.1332×10^{-6}	86.90 days
3	16	Edge switch	0.6665×10^{-6}	17.37 days
2	4	PSU	0.3999×10^{-6}	28.94 days
1	1	Compute nodes	0.1757×10^{-4}	15.8 hours

In the existing asynchronous flush implementation, an additional process runs on each compute node to read data from node-local storage and to write data to the PFS. Even though this process self-throttles its run time, it uses CPU time and other resources that can impact the application. Figure 1 shows the run time of the IOR benchmark [7] using SCR with synchronous flush, asynchronous flush, and no flush on the LLNL Sierra system, a 1944-node Linux cluster with 12 cores per node. The result shows, if we consume a core to conduct asynchronous checkpointing, then we contend with and slow down the application. The net cost to the application when it uses all 12 cores is similar to the cost of simply halting the application until it finishes writing a synchronous checkpoint to the PFS. Motivated by this result, in this work we develop a lighter weight asynchronous checkpoint method to minimize the impact on application run time.

III. Resiliency on a peta-scale supercomputer

We analyzed the failure history of TSUBAME2.0 [2] during the period of November 1, 2010 to April 6, 2012 to obtain its failure rate. In Table I, we show the categories of failures that occurred. The failures are ranked highest to lowest in terms of how many compute nodes are affected by a failure in a particular category. For example, failures of the PFS are at category 5, because the PFS is shared by all compute nodes; if it fails, all running processes that access the PFS fail.

Category 5 failures include failures of the PFS and core switches. *Core switches* are upper level switches, which connect lower level *edge switches*. A core switch is redundantly connected to edge switches in order to ensure quality of service. On a core switch failure, the running job using that switch fails. However, routing is updated and later jobs

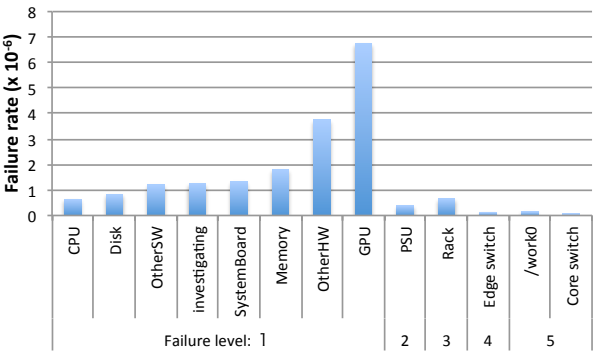


Fig. 2: Break-down of failure rates

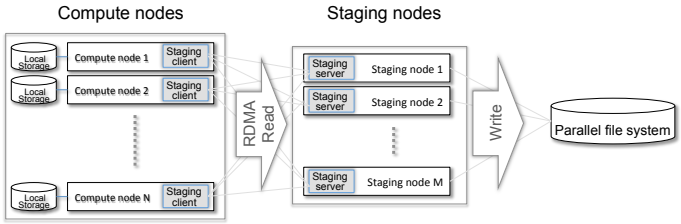


Fig. 3: The non-blocking checkpointing system

can communicate across processes without it. We categorize rack failures at level 4. These failures affect 32 nodes at a time. Rack failures are usually caused by a faulty water cooling unit sensor. Because the water cooling unit sensors are independent from computing components, running jobs do not fail immediately. However, a job running on the rack will fail at some point, assuming the faulty sensor is not fixed. Failure categories 3, 2, and 1 include edge switch, PSU (power supply unit) and compute node failures. A category 3 component is shared by 4 category 2 components, which are connected to 4 compute nodes. Thus, on a category 2 failure, 4 compute nodes fail, and on a category 3 failure, 16 (4x4) compute nodes concurrently fail.

Most failures are in category 1, affecting only single node. The rate is two orders of magnitude larger than the rates of the other categories. Because compute nodes are becoming more complex with increasing computational power, node failures are becoming more frequent. Figure 2 shows failure rates of each component on a compute node; approximately half of the failures on a compute node arise from GPU failures. On TSUBAME2.0, the GPU usage is high, as more applications use the GPUs through CUDA, OpenCL or OpenACC. This burdens GPUs and increases the overall temperature, which results in high failure rate of GPUs.

Although failures can disable one or more nodes, our checkpoint/restart system can often mitigate them by caching redundant copies of checkpoints on the compute nodes. For example, on a PFS failure, node-local checkpoints are not lost and the failed job can recover using the cached checkpoints. In addition, HPC systems usually manage a redundant PFS, e.g., TSUBAME2.0 has 1 Lustre and 1 GPFS file system and LLNL clusters have multiple Lustre file systems. Thus, we can restart the failed job by recovering from node-local checkpoints and using a different PFS for checkpoint storage. Similarly, we can use node-local checkpoints to recover from core and edge switch failures. For failures that disable the compute node itself, the job can be restarted using redundant copies of the checkpoints that are cached on non-failed compute nodes.

The most efficient redundancy scheme applied by our system in terms of size is XOR. With XOR, small sets of nodes collectively compute and store redundancy data, similarly to RAID-5 [8], [9]. XOR withstands node failures as long as two or more nodes from the same set do not fail concurrently. Since most failures affect a single node at a time, these failures can be recovered fairly reliably from an XOR checkpoint. With multi-level checkpointing, the role of PFS checkpoints shifts from primary to secondary checkpoint storage.

IV. Non-blocking Checkpointing System

Overlapping I/O with computation by delegating operations to dedicated I/O nodes improves application performance and mitigates PFS workload [10], [11]. Further, multi-level checkpointing reduces the frequency of PFS checkpoints. By combining long intervals between consecutive PFS checkpoints with asynchronous flushes (i.e., *non-blocking checkpoints*), we can copy data to the PFS at a slow rate to reduce the impact on the application as well as the PFS. This section details the design of our non-blocking checkpointing system.

A. Architecture

As Figure 3 shows, our non-blocking checkpointing system has two types of nodes: *compute nodes* and *staging nodes*. The compute nodes are the nodes on which the application executes. The staging nodes are a group of nodes that we use to transfer checkpoints from the compute nodes to the PFS. The staging nodes asynchronously read checkpoint data from the compute nodes and write data to the PFS while the application continues to execute and to write node-local checkpoints. Generally, each staging node handles multiple compute nodes; we determine the exact ratio through modeling and experimental testing (See Sections V and VI).

A *staging client* process runs on each compute node, and a *staging server* process runs on each staging node. When SCR finishes caching a checkpoint (node-local checkpoint) that is to be flushed to the PFS, it signals the staging client process via a library function call. The staging client then sends a request to the staging server and the two processes execute a protocol to transfer the checkpoint; details appear in Section IV-B. The staging server reads checkpoints from the compute nodes using Remote Direct Memory Access (RDMA) to minimize CPU usage on the compute nodes.

Using additional nodes to transfer checkpoint data with RDMA provides our non-blocking checkpointing system with two advantages. First, it minimizes compute node CPU usage. On a flush operation, even asynchronous checkpointing can impact application runtime as Figure 1 shows. With RDMA,

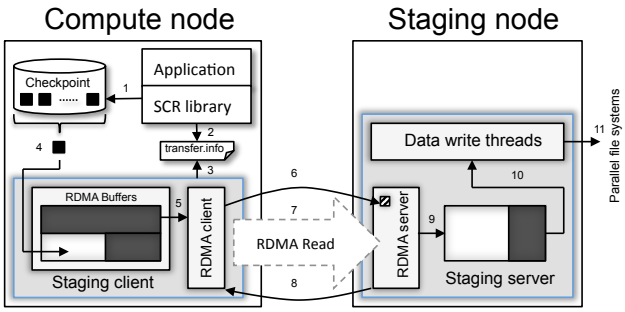


Fig. 4: Non-blocking checkpointing client/server using RDMA

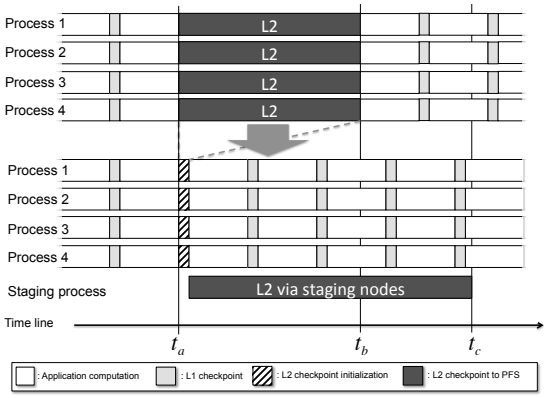


Fig. 5: Non-blocking checkpointing can hide L2 checkpoint overhead

the asynchronous checkpointing system drains a checkpoint from compute nodes to the PFS while minimizing the impact on application runtime. Because staging nodes are independent from compute nodes, we can coordinate between the staging nodes to throttle the RDMA read rate without impacting the performance of the running application.

Second, staging nodes can support balancing overall data center I/O. A PFS is often a shared resource. Ill-timed I/O patterns between two applications accessing the PFS can reduce the performance of both applications, which is particularly likely with checkpointing since it is one of the most I/O intensive operations. Staging nodes write checkpoints to the PFS independently of compute node activities, which allows us to throttle I/O to the PFS without directly throttling the application I/O rate on the compute nodes. This paper focuses on minimizing CPU usage; we leave I/O throttling techniques for optimizing overall data center I/O as future work.

B. RDMA Checkpoint Transfers

We implement an RDMA transfer system for asynchronous checkpointing based on the SCR library [6]. The existing SCR asynchronous flush implementation executes an extra process on each compute node, which reads a checkpoint from local storage and directly writes that checkpoint to the PFS. This extra process does substantial work on the compute node, and so it contends with and slows down the application. In contrast, our staging client process that runs on the compute node does minimal work, and most of the effort is delegated to the staging server process on the staging node. Other checkpoint management, such as versioning, checkpoint location, and redundancy scheme, relies on the original SCR library.

Figure 4 illustrates the architecture of our design with an example. First, assume that SCR has cached a checkpoint in local storage (Step 1). After applying its redundancy scheme to this checkpoint, SCR writes information into a file called *transfer.info* requesting that the checkpoint be copied to the PFS (Step 2). Among other information, the transfer.info file includes the source path of the checkpoint files in local storage as well as the destination paths to which the files should be written on the PFS.

The staging client periodically checks the transfer.info file for requests, and sleeps for the rest of the time. Thus, our design does minimal work on the compute node (Step 3). If the staging client detects a new request, it reads the checkpoint file from the source path and copies the data to a local RDMA buffer (Step 4). Once the staging client fills the buffer, it calls an RDMA client function to initiate the data transfer (Step 5). Since the RDMA client function returns control immediately, the staging client can read the next chunk to one of the buffer entries in the buffer pool (Figure 4 shows the double-buffering case) while the RDMA client transfers checkpoint chunks to the staging server. The RDMA client sends a short message containing details about the checkpoint and the RDMA buffer to the RDMA server (Step 6). When the RDMA server receives this message, it issues an RDMA read request to read a chunk of the remote buffer space into a local buffer (Step 7). Then it sends an acknowledgment message to the RDMA client (Step 8), and it copies the data to the staging server buffer (Step 9). This protocol continues until all checkpoint data has been copied from the staging client to the staging server. Finally, data writer threads write the checkpoint to the PFS in parallel with RDMA reads by the RDMA server (Steps 10 and 11).

An RDMA operation can only read or write remote memory regions of a few MB of data. Thus, we divide a checkpoint into smaller chunks, which the RDMA server remotely reads one by one. To reduce the number of staging nodes, a transfer server can concurrently handle RDMA requests from multiple transfer clients. However, a large amount of incoming checkpoint data can cause buffer overflow on a staging node. Thus, our staging client, which runs on the compute node, reads a small chunk of data from the compute node storage to a registered memory region for RDMA. The staging server then pulls the chunk (*incoming*) region to its own space and writes to the PFS (*outgoing*). If buffered checkpoint data on a staging node exceeds a specified buffer size limit, the staging server throttles the RDMA read rate (*incoming*) to balance incoming and outgoing checkpoint data. To avoid imbalance in incoming and outgoing data, we determine the number of staging nodes according to outgoing PFS throughput.

C. Blocking and Non-blocking Checkpointing

Figure 5 shows the difference between blocking and non-blocking checkpointing. To clarify the differences, we characterize both schemes with two metrics, *checkpoint overhead* and *checkpoint latency*. Checkpoint overhead (C) is the increased execution time of an application because of checkpointing. Checkpoint latency (L) is the time to complete a checkpoint.

During blocking checkpointing, each process writes its checkpoint data to the PFS, and blocks until the checkpoint operation completes. Thus, the checkpoint overhead is identical to the checkpoint latency, i.e., $C_{blk} = L_{blk} = t_b - t_a$. N iterations of blocking checkpointing increase application run time by $N \times C_{blk}$.

With non-blocking checkpointing, each process continues computation during the PFS checkpoint so checkpoint overhead ($_{nblk}$) is generally smaller than checkpoint latency ($L_{nblk} = t_c - t_a$). Application characteristics determine C_{nblk} and L_{nblk}. If an application is computation or network bound, C_{nblk} and L_{nblk} increase due to resource contention, and L_{nblk} can become larger than L_{blk}. To initiate a non-blocking checkpoint, the application must write its checkpoint data to local storage. During the write operations, the application is blocked so we add this overhead to C_{nblk}.

Non-blocking checkpointing has advantages over blocking checkpointing. We can minimize C_{nblk} by slowly writing checkpoint data to the PFS, thereby alleviating resource contention. Because lower-level checkpoints can continue to be cached on the compute nodes during a non-blocking checkpoint, the application can take more frequent checkpoints and increase resiliency with low checkpoint overhead. In contrast, when an application takes a blocking PFS checkpoint, the application loses C_{blk} potential computation time and it is significantly more vulnerable to failure, as heavy PFS load increases the likelihood of PFS failures.

Thus, we intuitively expect non-blocking checkpointing to be more efficient than blocking checkpointing. However, non-blocking checkpointing has a disadvantage. In Figure 5, the blocking checkpoint completes at t_b while the non-blocking checkpoint finishes at t_c. If a failure that requires a PFS checkpoint occurs in the period between t_b and t_c, a non-blocking checkpointing system incurs a catastrophic rollback to an older checkpoint. Alternatively, blocking checkpointing only rollbacks to t_b. Therefore, with non-blocking checkpointing, the checkpoint interval, C_{nblk}, L_{nblk}, and the frequency of each level of checkpoint must be optimized to lower the risk of the catastrophic rollback.

V. Non-blocking Checkpointing Model

As mentioned previously, with non-blocking checkpointing, several factors are critical to performance: checkpoint interval, C_{nblk}, L_{nblk}, and frequency of each level of checkpoint. To determine the optimal values, we extend an existing model of a multi-level checkpointing system [4] to support our non-blocking checkpointing system.

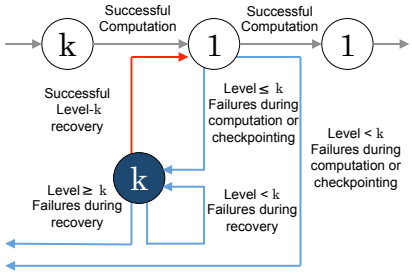


Fig. 6: The basic structure of the non-blocking checkpointing model

A. Assumptions

For simplicity, we make several assumptions in our non-blocking checkpointing model. Because we build on an existing model, we include that model's assumptions [4]. We highlight the important assumptions here.

We assume that failures are independent across components and occur following a Poisson distribution. Thus, a failure within a job does not increase the probability of successive failures. In reality, some failures can be correlated. For example, failure of a PSU can take out multiple nodes. However, topology-aware techniques can provide very low probability of those failures affecting processes in the same XOR set, which eliminates the need to restart from the PFS. SCR also excludes problematic nodes from restarted runs. Thus, the assumption implies that the average failure rates do not change and dynamic checkpoint interval adjustment is not required during application execution.

We also assume that the costs to write and to read checkpoints are constant throughout job execution. In reality, I/O performance can fluctuate because of contention for shared PFS resources. However, staging nodes serve as a buffer between the compute nodes and the PFS. Thus, our system mitigates PFS performance variability.

If a failure occurs during non-blocking checkpointing, we assume that checkpoints cached on failed nodes have not been written to the PFS. Thus, we must recover the lost checkpoint data from redundant stores on the compute nodes, if possible, and if not, locate an older checkpoint to restart the application. We can use either an older checkpoint cached on compute nodes, assuming multiple checkpoints are cached, or a checkpoint on the PFS.

B. Basic model structure

As in the existing model [4], we use a Markov model to describe run time states of an application. We construct the model by combining the basic structures that Figure 6 shows. The basic structure has *computation* (white circle) and *recovery* (blue circle) states labeled by a checkpoint level. The computation states represent periods of application computation followed by a checkpoint at the labeled level. The recovery state represents the period of restoring from a checkpoint at the labeled level.

If no failures occur during a compute state, the application transitions to the next right compute state (gray arrow). We denote the checkpoint interval between checkpoints as t, the cost of a level c checkpoint as c_c, and rate of failure requiring level k checkpoint as λ_k. The probability of transitioning to the next right compute state and the expected time before transition are $p_0(t + c_c)$ and $t_0(t + c_c)$ where:

$$\begin{aligned} p_0(T) &= e^{-\lambda T} \\ t_0(T) &= T \end{aligned}$$

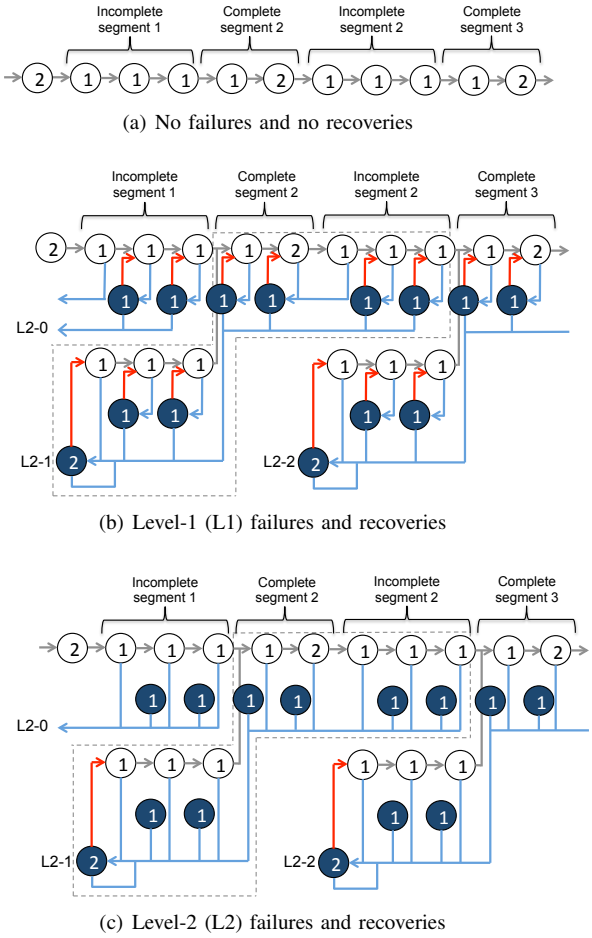
We denote λ as the summation of all levels of failure rates, i.e., $\lambda = \sum_{i=1}^{L} \lambda_i$ where L represents the highest checkpoint level. If a failure occurs during a compute state, the application transitions to the most recent recovery state that can handle the failure (blue arrow). If the failure requires a level i checkpoint or less to recover and the most recent recover state is at level k where $i \leq k$, the application transitions to the level k recovery state. The expected probability of a level i failure in an interval $t+c_c$, and the run time before the transition from the compute state c to the recovery state k are $p_i(t+c_c)$ and $t_i(t+c_c)$ where:

$$\begin{aligned} p_i(T) &= \frac{\lambda_i}{\lambda}(1 - e^{-\lambda T}) \\ t_i(T) &= \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})} \end{aligned}$$

During recovery, if no failures occur, the application transitions to the compute state at the restored checkpoint. If the recovery cost from a level k checkpoint is r_k, the expected probability of the transition and the expected run time are $p_0(r_k)$ and $t_0(r_k)$. If a failure requiring a level i checkpoint occurs while recovering, and $i < k$, we assume the current recovery state can retry the recovery. However, if $i \geq k$, we assume the application must transition to a higher-level recovery state. The expected probabilities and times of failure during recovery are $p_i(r_k)$ and $t_i(r_k)$. We also assume that the highest level recovery state (level L), which uses checkpoints on the PFS, can be restarted in the event of any failure $i \leq L$.

C. Non-blocking checkpoint model

Our model of non-blocking checkpointing combines the basic structures from Figure 6. Figure 7 shows a two level example. If no failures occur during execution, the application transitions across the compute states in sequence (Figure 7(a)). In this example, level 1 (L1) checkpoints (e.g., XOR checkpoints) are blocking and level 2 (L2) checkpoints (e.g., PFS checkpoints) are non-blocking. With blocking checkpointing, the checkpoint is available when the corresponding compute state completes. Thus, if an L1 failure occurs, the application transitions to the most recent L1 recovery state (Figure 7(b)). Alternatively, with non-blocking checkpointing, the L2 checkpoint is not finished when the L2 compute state completes. Thus, we divide compute states into two segment types: *incomplete segments* and *complete segments*. A computation state in an incomplete segment represents a period when the L2 checkpoint has started but is not yet completed. For example, if an L2 failure occurs in incomplete segment 1, the application transitions to the recovery state for the last completed L2 checkpoint (L2-0 in Figure 7(c)). When an L2 failure occurs in a complete segment 2, the application transitions to the recovery state for the completed L2 checkpoint (L2-1 in Figure 7(c)).



(a) No failures and no recoveries

(b) Level-1 (L1) failures and recoveries

(c) Level-2 (L2) failures and recoveries

Fig. 7: Transition diagram of the non-blocking checkpointing

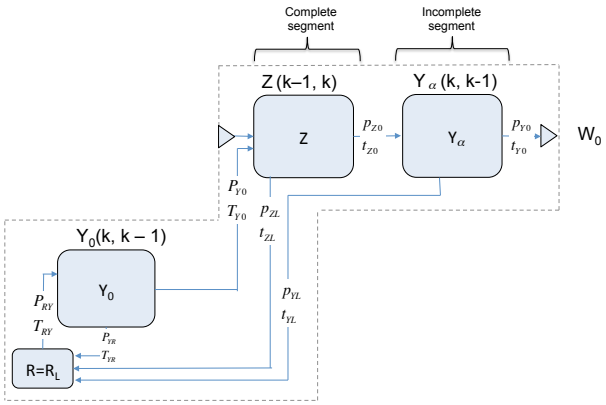


Fig. 8: General non-blocking model structure: $W(k)$ state

Interference from other asynchronous operations can inflate the time in compute states before transitions. An overhead factor, α, quantifies the overhead on compute states in incom-

plete segments. We define the time spent in compute states in an incomplete segment as $(1+\alpha)t$, where t is the sum of the computation time and the time to complete a checkpoint. Thus, the expected probability and time for compute states in an incomplete segment, become $p_0(T)$, $t_0(T)$, $p_i(T)$ and $t_i(T)$ where $T = (1+\alpha)\,t + c_c$.

Throughout our evaluation (Section VI), we employ two-level checkpointing, but our model can be extended to an arbitrary N-level checkpointing model. Figure 8 shows the general structure of our non-blocking checkpointing model. Our model is composed of hierarchical states, with the outermost state denoted as a $W(k)$ $state$. The definition of $Z(k,c)$ and its expected probabilities and times for transition to other states were defined in the existing model [12]. To distinguish compute states in complete and incomplete segments, we extend the definition of $Y(k,c)$ to reflect the interference as $Y_x(k,c)$, where x denotes the overhead factor. Using the model, we can compute the *expected time* to complete a given number of compute states with an arbitrary number of checkpointing levels.

VI. Evaluation

This section compares our non-blocking checkpointing system to the existing SCR implementation [4]. For illustration, we model a two-level system in which the first level uses SSD on the compute nodes with a RAID-5-like redundancy scheme and the second level is a PFS.

A. Tuning of Non-blocking Checkpointing

Checkpoint efficiency highly depends on I/O throughput so we must tune I/O operations such that staging nodes fully exploit the available I/O performance. Generally, we can increase I/O throughput to a PFS by writing with multiple threads, so we multi-thread our staging server process. Our goal is to find the optimal numbers of threads and staging nodes such that we can obtain near peak PFS performance. As a target PFS, we use the Lustre file system [13] on TSUBAME2.0. A TSUBAME2.0 node has two sockets of Intel Xeon X5670, 58GB of DDR3 1333MHz memory, 120GB of local SSD and three Tesla M2050 GPUs. The nodes are connected through Dual-Rail QDR Infiniband(x4).

Figure 9 shows the write throughput of one staging node with different numbers of *data writer threads*. We achieve the highest performance (1.6 GB/s) on a single staging node with 16 threads. We then explore how many staging nodes can exploit the Lustre file system. Figure 10 presents aggregate write throughput with different staging node counts, each using 16 data writers. Aggregate write throughput rapidly increases from 1 to 32 staging nodes but then quickly saturates around 8 GB/s beyond 32 staging nodes. Thus, we use 32 staging nodes and set the staging server to run with 16 data writer threads. Under this set up, checkpoint data can be transferred to the PFS at a rate of 6.4 GB/s via 32 staging nodes, which is only 2.3% of the 1408 TSUBAME2.0 thin nodes.

Whenever the staging client and server processes read checkpoint data from compute nodes in the background, significant overhead is added to the application runtime due to resource contention. The amount of overhead depends on the read rate. To estimate this overhead, we transferred checkpoints while running the Himeno benchmark [14] as a target application. This benchmark solves Poisson's equation using the Jacobi iteration method. The Himeno benchmark is a stencil application in which each grid point is repeatedly updated using only neighbor points in a domain. This computational pattern frequently appears in numerical simulation codes for solving partial differential equations. Many fluid dynamics phenomena can be described by partial differential equations over multi-dimensional Cartesian grids, including weather, seismic waves, heat flow, and electric charge and magnetic field distribution in a domain.

Figure 11 shows the overhead factor imposed on the Himeno benchmark while varying the checkpoint read rate of a staging node. We find that the overhead factor roughly increases linearly with the read rate. Based on the result, we model the overhead factor (α) of the Himeno benchmark as $\alpha = cx$ where c is 0.008768, and x represents the checkpoint read rate of a staging node in GB/s. The parameter c is derived from the slope of fitting a line to the data in Figure 11. With 32 staging nodes, we calculate the read rate per staging node to be 209.5 MB/s, which is derived by dividing the aggregate write throughput when using 32 nodes in Figure 10 by 32, the number of staging nodes. Thus, the overhead factor model gives us the overhead factor of 0.00184 ($= 0.008768 \times 0.20905$). We add this overhead to the cost of computation in our model when we compute efficiency of asynchronous checkpointing.

SCR can adjust the degree of resiliency by changing the number of processes in each XOR group used to compute redundancy data. Figure 12 shows encoding throughput for different XOR group sizes. Resiliency improves with smaller XOR groups, but XOR encoding throughput decreases. For an XOR checkpoint, SCR computes the parity of each block as in RAID-5 [8], [9], which creates $S = B + \frac{B}{N-1}$ bytes of encoded checkpoint data from B bytes of original checkpoint data within N members of an XOR group. Since the encoding time increases linearly with the encoded checkpoint size, the large XOR group size, NS, saturates the XOR encoding rate. As Section III showed, most failures affect just one node. Thus, we use XOR checkpoints only to handle failure category 1 in Table I, and we handle the other failure categories $k = 2, 3, 4 \ldots 5$ by a PFS checkpoint. Thus, we set the XOR encoding rate as the saturated maximal rate, 400MB/s.

B. Efficiency Comparison

As future systems will be larger and will have larger memory sizes, failure rates and checkpoint size are expected to increase. To explore the effects, we increase failure rates and checkpoint costs by factors of 1, 2, and 10, and compare efficiency between blocking and non-blocking checkpointing. We use 29 GB for the checkpoint size per compute node, which is half of the memory of a TSUBAME2.0 thin node. As Figure 12 shows, an XOR encoding rate is constant regardless of the number of compute nodes, which means XOR encoding

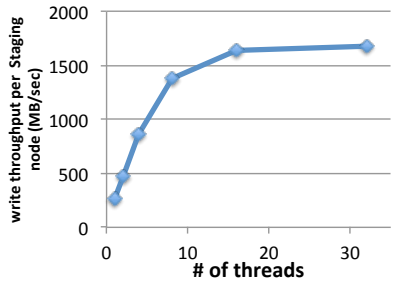


Fig. 9: Impact of varying data writer count

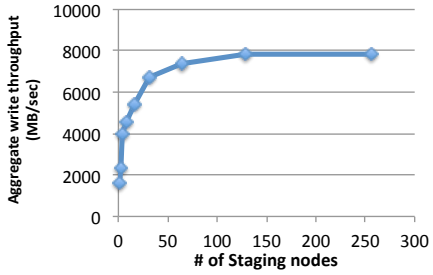


Fig. 10: Aggregate write throughputs

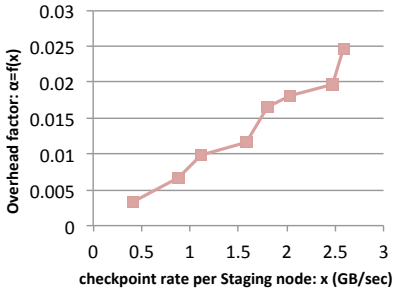


Fig. 11: Empirical overhead factor

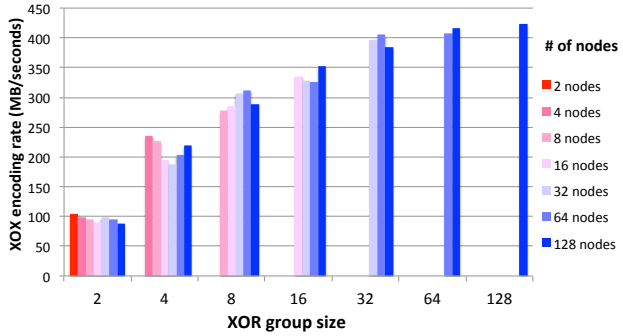


Fig. 12: XOR encoding performance per node

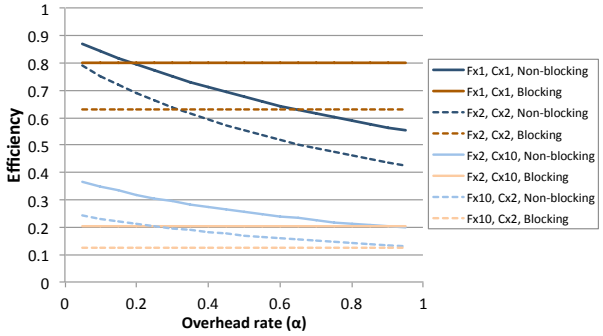


Fig. 14: Efficiency under varying the overhead factor: α

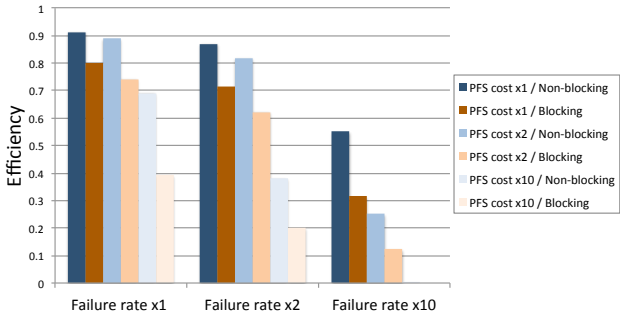


Fig. 13: Efficiency of blocking and non-blocking checkpointing

scales with system size. Thus, when we increase checkpoint costs, we increase only PFS checkpoint cost.

Figure 13 shows the *efficiency* of both checkpointing methods under different failure rates and checkpoint costs. We define the *efficiency* as $\frac{ideal\_time}{expected\_time}$. The *ideal_time* is the run time if the application encounters no failures and takes no checkpoints, while *expected_time* is the expected run time computed from our model for non-blocking checkpointing and the original model [4] for blocking checkpointing. When we compute efficiency, we optimize Level 1 counts between Level 2 checkpoints, and the interval between checkpoints, given failure rates and checkpoint costs, which yields the *maximal* efficiency. The non-blocking method always achieves higher efficiency than the blocking method. The efficiency gap becomes more apparent with higher failure rates and higher checkpoint cost. This is because the long time to take a PFS checkpoint during blocking checkpointing increases the likelihood of a lower level failure occuring during the PFS checkpoint, so the application must rollback to the beginning. However, with non-blocking checkpointing, the application can rollback to the most recent XOR checkpoint. Further, since overhead of a blocking checkpoint is identical to checkpoint latency, which is directly added to application run time, the efficiency decreases more quickly than with non-blocking checkpointing.

Since staging nodes write checkpoints to the PFS independently of compute node activities, they can throttle their write rate without reducing application performance. We found that we could read checkpoints from compute nodes at half of the maximum bandwidth (i.e., PFS cost $\times 2$), but still maintain 90% efficiency with current failure rates.

Because a non-blocking checkpoint overlaps with application computation, non-blocking checkpointing can impact the application run time depending on the overhead factor, α, in different applications. If the overhead factor increases enough, our non-blocking checkpointing could be less efficient than blocking checkpointing. Figure 14 shows efficiency with increasing overhead factor and different failure rates and PFS checkpoint costs. F and C denote the base failure rate and PFS checkpoint cost. Blocking checkpointing can become more efficient than non-blocking with a larger overhead factor at current failure rates and cost. However, in future systems where

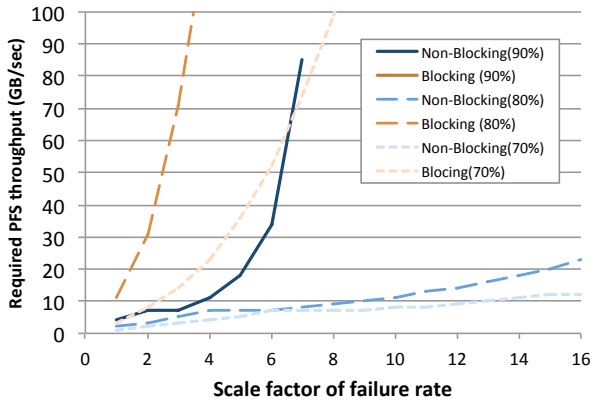


Fig. 15: Required PFS throughput at different failure rates

the failure rates and cost increase, non-blocking checkpointing can be effective even with a large overhead factor. With large failure rates and checkpointing costs, the checkpointing interval decreases so that checkpointing overhead dominates the overall run time. Since an application is blocked with blocking checkpointing, the checkpoint latency impacts application run time more than with non-blocking checkpointing in future systems. As for different systems, we observed that three clusters at LLNL show similar failure rates [4], where system failures can be recovered from level 1 checkpoints and frequent level 2 checkpoints are not required. Thus, non-blocking checkpointing would benefit other systems.

C. Building an Efficient and Resilient System

When building a reliable data center or supercomputer, two major concerns are cost of the PFS and the PFS throughput required to maintain high efficiency. Generally, we want to minimize cost, but not sacrifice performance. Using our model, we can predict the required PFS bandwidth to achieve high system efficiency when using our checkpointing system.

Figure 15 shows the PFS bandwidth required to maintain 90%, 80%, and 70% efficiency under increasing failure rates, scaled from 1× up to 16× today's rates. Overall, our checkpointing system outperforms blocking checkpointing. Failure rates can increase by 16× and still require modest PFS throughputs for 80% application efficiency. However, at 90% efficiency, the bandwidth requirement rises sharply with failure rates larger than 5×. Here, as failure rates increase, the optimal checkpoint interval decreases, increasing the overhead contribution of the L1 checkpoints. As the overhead of L1 checkpoints approaches 10%, the allowed overhead for writing PFS checkpoints approaches 0%. Therefore, achieving 90% efficiency at higher failure rates requires throughput values approaching infinity. The curve for blocking checkpointing at 90% efficiency follows the same trend, but is much more severe, so much so that it does not appear in the figure. These trends imply that reducing the overhead of L1 checkpoints will be necessary on machines with higher failure rates.

With blocking checkpointing, systems require higher PFS throughput to minimize the risk of failure during a PFS checkpoint. Further, blocking checkpointing uses the PFS only when taking a PFS checkpointing, which means the PFS underutilized during most of the application run. With our checkpointing system, we not only hide PFS checkpoint overhead, but use PFS throughout application execution.

VII. Related Work

Multi-level checkpointing [4], [5] is a promising technique for fault-tolerant execution. Bautista-Gomez et al. [5] proposed multi-level checkpointing using local SSDs and a PFS. They use Reed-Solomon (RS) encoding for highly resilient cached checkpoints to reduce PFS usage. Generally, PFS usage is costly when compared to local storage, and the PFS is accessed less often in multi-level checkpointing. However, increasing failure rates require checkpoints to a PFS more frequently. Thus, even with multi-level checkpointing, checkpointing to a PFS is crucial for future systems.

Asynchronous I/O has long been used to hide I/O bottlenecks [11], [15]–[17]. These techniques enable applications to parallelize I/O with computation to increase CPU utilization and to enhance I/O performance. Patrick et al. [15] presented a comprehensive study of different techniques of overlapping I/O, communication, and computation, and showed the performance benefits of asynchronous I/O. Nawab et al. [16] asynchronously transfer multiple striped TCP data streams to increase I/O performance in Grid environments. An asynchronous staging service using RDMA proposed by Hasan et al. [11] is the closest research to ours. The authors achieved high I/O throughput by using additional nodes. As we observed, optimizing performance requires determination of the proper number of staging nodes for a given number of compute nodes. However, the comprehensive study on the problem is not shown nor do they present their solution. To deal with bursty I/O requests, Liu et al. [17] simulated a system design that integrates SSD storage on I/O nodes as burst buffers. They found that such a system could deliver high I/O bandwidth to an application while reducing the demands on the parallel file system. In this work, we effectively implement a system with burst buffers on the compute nodes, and through experiment and modeling, we arrive at the same conclusions regarding the need for and benefits of asynchronous I/O to the parallel file system.

Optimization of checkpoint interval is critical, because checkpointing is an expensive operation. Several optimization techniques have been studied. Young [18] proposed a method to determine the optimal checkpoint interval for single-level, blocking checkpoints. Vaidya extended the model to support non-blocking checkpoints, called *forked checkpoints* [19], and two-level checkpoints [20]. Vaidya also combined both methods to support a two-level non-blocking checkpoint system to achieve higher efficiency [21]. Vaidya's model assumes that at most one fast-level checkpoint is taken between each slow-level checkpoint. However, in current multi-level checkpointing systems, the fastest checkpoints, which are often saved to node-local storage, are orders of magnitude faster than the

slowest checkpoints saved to the PFS. To account for this, we extend prior work to model an arbitrary number of node-local checkpoints between consecutive PFS checkpoints.

VIII. Conclusion

We have designed and modeled a non-blocking checkpointing system that extends an existing multi-level checkpointing system. Our non-blocking checkpointing system enables applications to save checkpoints to fast, scalable storage located on the compute nodes and then continue with their execution while dedicated staging nodes copy the checkpoint to the PFS in the background. This capability simultaneously increases system efficiency and decreases required PFS bandwidth. Since applications spend less time in defensive I/O, we find that our non-blocking checkpointing system can improve machine efficiency by 1.1 to 2.0 times on future systems. Further, our model predicts that non-blocking checkpointing significantly reduces the PFS bandwidth required to maintain application efficiency. For example, to maintain 80% efficiency at $4\times$ today's failure rates, the PFS bandwidth required for non-blocking checkpointing is an order of magnitude less than that required by blocking checkpointing. Additionally, our non-blocking checkpointing system can maintain 80% efficiency with only modest PFS bandwidth requirements even when failure rates are $16\times$ higher than on current petascale systems.

Acknowledgment

This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes (LLNL-CONF-554431). This work was also supported by Grant-in-Aid for Research Fellow of the Japan Society for the Promotion of Science (JSPS Fellows) 24008253, and Grant-in-Aid for Scientific Research S 23220003.

References

[1] "TOP500 Supercomputing Sites," http://www.top500.org/.

[2] "TSUBAME 2.0 - Monitoring Portal," http://mon.g.gsic.titech.ac.jp/.

[3] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, pp. 012 022+, Jul. 2007. [Online]. Available: http://dx.doi.org/10.1088/1742-6596/78/1/012022

[4] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, Nov. 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[5] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WS, USA, 2011.

[6] "Scalable Checkpoint / Restart Library," http://sourceforge.net/projects/scalablecr/.

[7] "IOR HPC Benchmark," http://sourceforge.net/projects/ior-sio/.

[8] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, 1988.

[9] W. Gropp, R. Ross, and N. Miller, "Providing Efficient I/O Redundancy in MPI Environments," in *Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting*, 2004.

[10] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1145/1362622.1362636

[11] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: Scalable Data Staging Services for Petascale Applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 39–48. [Online]. Available: http://dx.doi.org/10.1145/1551609.1551618

[12] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Detailed Modeling, Design, and Evaluation of a Scalable Multi-level Checkpointing System," https://library- ext.llnl.gov, Lawrence Livermore National Laboratory, Tech. Rep., Jul. 2010.

[13] "Lustre: A Scalable, High-Performance File System," http://wiki.lustre.org/index.php/Main_Page.

[14] R. Himeno, "Himeno benchmark," http://accc.riken.jp/HPC_e/himenobmt_e.html.

[15] C. M. Patrick, S. Son, and M. Kandemir, "Comparative Evaluation of Overlap Strategies with Study of I/O Overlap in MPI-IO," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 43–49, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1145/1453775.1453784

[16] N. Ali and M. Lauria, "Improving the Performance of Remote I/O Using Asynchronous Primitives," pp. 218–228. [Online]. Available: http://dx.doi.org/10.1109/HPDC.2006.1652153

[17] N. Liu, C. Jason, C. Philip, C. Christopher, R. Robert, G. Gary, C. Adam, and M. Carlos, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *MSST/SNAPI*, Apr. 2012.

[18] J. W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Commun. ACM*, vol. 17, pp. 530–531, Sep. 1974. [Online]. Available: http://dx.doi.org/10.1145/361147.361115

[19] N. H. Vaidya, "On Checkpoint Latency," College Station, TX, USA, Tech. Rep., 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=892900

[20] ——, "A Case for Two-Level Distributed Recovery Schemes," *SIGMETRICS Perform. Eval. Rev.*, vol. 23, no. 1, pp. 64–73, May 1995. [Online]. Available: http://dx.doi.org/10.1145/223586.223596

[21] ——, "Another Two-Level Failure Recovery Scheme," College Station, TX, USA, Tech. Rep., 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=892923